Sorting and Hardware Assisted Rendering for Volume Visualization

Clifford Stein
Barry Becker
Nelson Max

Lawrence Livermore National Laboratory
Livermore, CA 94551 U.S. A.

Abstract

We present some techniques for volume rendering unstructured data. Colors and opacities are interpolated between vertices using hardware assisted texture mapping. We also present an O(n^2) method for sorting **n** *arbitrarily shaped convex polyhedra prior to visualization. It generalizes the Newell, Newell and Sancha sort for polygons to 3-D volume elements.*

Introduction

This project grew out of the need to visualize unstructured meshed vector fields such as those found in existing finite element modeling code. Some volume rendering applications do not require more than one color. However, we have developed a visualization tool for rendering multi-colored elements, such as colored flow volumes in a vector field, using an implementation of the Shirley-Tuchman [1] algorithm. While monochromatic elements can be composited in any order as shown by [2], data sets containing many colors must be composited in either a back-to-front or front-to-back order. Indeed, most volume rendering applications have color and opacity variations which require sorting.

This paper describes the visualization tool we have developed which can display a set of convex, non-intersecting polyhedra, with unique colors and opacities assigned to each vertex, using hardware assisted texture mapping. It uses an implementation of the Shirley-Tuchman algorithm to render the polyhedra once they have been subdivided into tetrahedra. Rendering colored volumes requires compositing the elements in a back to front order. Hence, we present an algorithm that sorts these unstructured elements before they are subdivided. The algorithm will correctly sort unstructured topologies of convex polyhedra that are devoid of intersections and cyclically overlapping polyhedra (see Figure 4).

Volume Rendering

We have developed a new volume rendering approximation which takes advantage of the texture mapping and compositing available on modern graphics workstations. This allows tetrahedra to be composited by the projected tetrahedra algorithm of Shirley and Tuchman, without artifacts due to linear approximation of the non-linear opacity effects.

The Shirley-Tuchman algorithm classifies the projection of each tetrahedron into one to four cases consisting of one to four triangles. Figure 1 shows the two non-degenerate cases, where no vertex projects onto another vertex or edge.

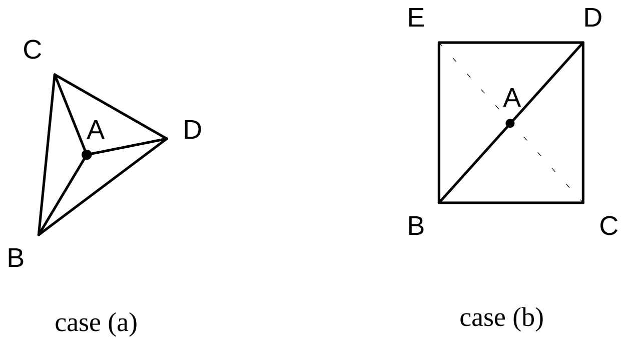


Figure 1.

A tetrahedron can project to three triangles as in case (a), or four triangles as in case (b). Each case has a single "thick" vertex A; the other vertices on the profile are called "thin". The thick vertex is the projection of the 3-D segment A_0A_1 where a viewing ray intersects the tetrahedron. In case (a), A_0 is A, and A_1 lies on the face BCD. In case (b), A_0 is on edge BD, and A_1 is on edge CE.

Assume that the color C(x) and extinction coefficient $\tau(x)$ vary linearly across the tetrahedron. Then these quantities can be interpolated across faces or edges to give values $\tau_0=\tau(A_0)$, $C=C(A_0)$, $\tau_1=\tau(A_1)$ and $C_1=C(A_1)$. Shirley and Tuchman show that the total opacity α of the segment A_0A_1 is $\alpha = 1 - \exp(-l(\tau_0 + \tau_1)/2)$, where l

is the length of the ray segment A_0A_1. The opacity at the profile vertices is zero. They approximate the color at the thick vertex to be $(C_0 + C_1)/2$; we will improve this.

For each triangle, the color and opacity are interpolated linearly from the three vertex values to the interior, usually along the edges and then across scan lines, as in Gouraud shading. Then the interpolated color C_i and opacity α_i are composited over the old pixel color C_{old} to give the new color C_{new}, by the formula: $C_{new} = \alpha_i C_i + (1-\alpha_i)C_{old}$. Often the linear interpolation and compositing steps can be performed by special purpose hardware available in the rendering engines of a graphics workstation.

The projected tetrahedra algorithm has several artifacts which produce incorrect colors, or Mach bands revealing the subdivision into tetrahedra. The first artifact comes from the linear interpolation of the color and opacity across the tetrahedra. This interpolation is not C^1 across the faces, and can produce Mach bands, particularly at faces which are parallel to the viewing direction and project to lines. The only cure is higher order interpolation, which is not available in hardware on most workstations.

However, there is a more serious problem with the algorithm, which occurs even when the color C and extinction coefficient τ are constant. The problem is easiest to understand when the color is zero, so that the image shows an opacity cloud hiding the background, and in 2-D, where the tetrahedra become triangles. Consider a strip of triangles T_0, T_1, T_2... of a constant width l as shown in Figure 2(a), projected vertically to a scan line. In triangle T_1, C is the "thick" vertex, where the opacity $\alpha = \exp(-\tau l)$, and α=0 at B and D. Figure 2(b) is a graph of the transparency $t_1(x) = 1 - \alpha_1(x)$ along the scan line, which is used to multiply the background color during compositing of triangle T_1. It is piecewise linear, because the opacity $\alpha(x)$ has been linearly interpolated across the scan line segments BC and CD. Similarly, Figure 2(c) shows the transparency $t_2(x)$ from triangle T_2. The final transparency along the segment CD, resulting from compositing both triangles on top of the background is the product $t(x)=t_1(x)t_2(x)$, shown as the quadratic polynomial segment above CD in Figure 2(d).

To derive the form of this quadratic polynomial, let x=sD+(1-s)C be the point a fraction $\boldsymbol{s}$ of the way from C to D.

Then

$$\begin{aligned} t_1(x) &= 1 - \alpha_1(x) \\ &= 1 - (s \cdot 0 + (1-s)\exp(-\tau l)) \\ &= 1 - (1-s)\exp(-\tau l) \end{aligned}$$

and similarly

$$t_2(x) = 1 - \alpha_2(x) = 1 - s \cdot \exp(-\tau l).$$

Thus

$$t(x) = t_1(x)t_2(x) = 1 - \exp(-\tau l) + s(1-s)(\exp(-\tau l))^2.$$

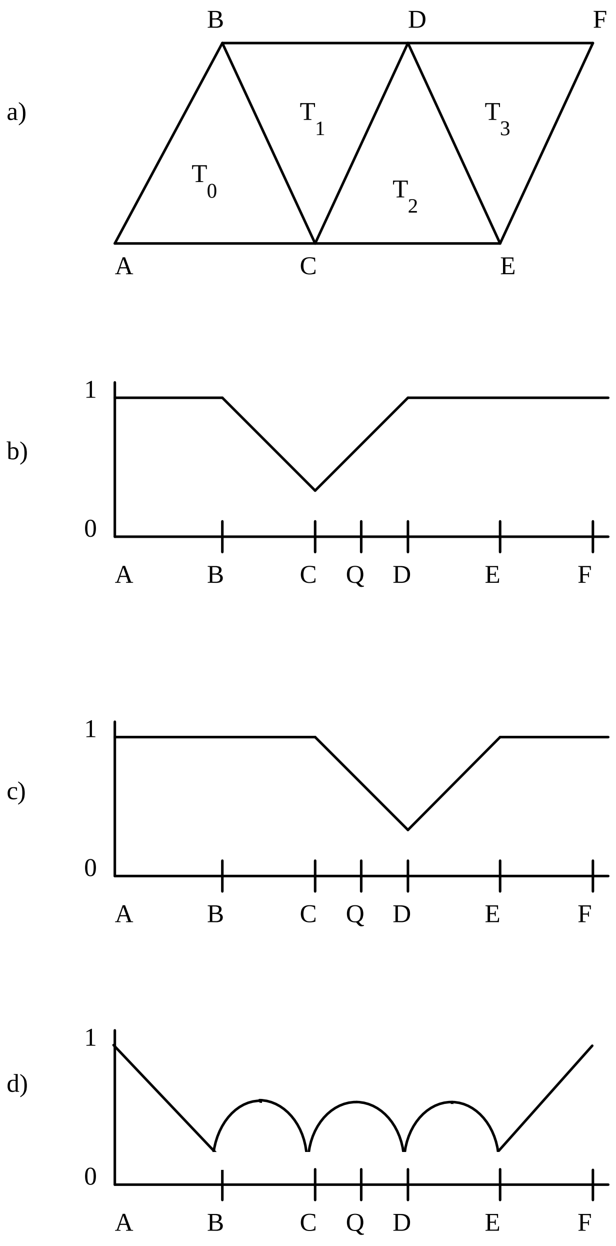


Figure 2

The transparency should actually be 1-exp(-τl), so the quadratic term $s(1-s)(\exp(-\tau l))^2$ represents the error due to approximating $t_1(x)$ and $t_2(x)$ linearly.

Other similar quadratic segments come from other projected diagonal edges, and the final intensity, proportional to the transparency if the background is uniform, is not C^1. In three dimensions, the corresponding effect produces Mach bands along the projections of edges of the tetrahedra.

The solution to this problem is to define $\alpha_1(x)$ correctly as 1-exp(-$s\tau l$). This requires a linear interpolation of the quantity τl, and then an exponential per pixel, which is not commonly available in hardware. Instead, we have used a texture map table on our SGI Onyx™ system. For the case of constant τ per tetrahedron, as in our flow volume application, we put the quantity $1-\exp(-u)$ in a one dimensional texture table, indexed by u [2]. The texture coordinate u was set to zero at the thin vertices of each triangle, and to τl at the thick vertex, and was interpolated by the shading hardware, before being used as an address to the texture table.

If τ varies linearly within each tetrahedron the product τl varies quadratically inside each triangle. Quadratic interpolation of texture coordinates was implemented in hardware on the Apollo DN10000VS [3], but was not available on our Onyx™. Therefore we used a 2-D texture table, with coordinates τ and l, and put $1-\exp(-\tau l)$ in the table.

Now consider the case when the color also varies linearly across the tetrahedron. The Shirley-Tuchman approximation $(C_0+C_1)/2$ for the color of the thick vertex is not precise; it weighs the two colors equally. The frontmost color should have greater weight, because the opacity along the ray segment hides the rear color more than the front one. Williams and Max [4] have found an exact formula for the color in this case, which they implement with the aid of table lookups. However, the supplementary arithmetic required goes far beyond what is practical in hardware computation at each pixel. As a compromise, we have used the exact analytic form of the color of the thick vertex, and then used the hardware to interpolate the color across each triangle. The colors of the thin vertices come from the original color specification, and the opacity is determined, as above, from a texture table. This compromise can be implemented entirely in hardware, and gives a fairly smooth color variation that seems to move appropriately when a colored volume density rotates.

Figures 7(a), (b), (d) and (e) all use texture mapping for the opacity. Figures 7(a) and (b) show a triangular prism divided into three tetrahedra. Figure 7(a) uses the average color $(C_0+C_1)/2$ at the thick vertices, while Figure 7(b) uses the more accurate color integration of Williams and Max [4]. Note that in Figure 7(b) the color of the yellow-orange vertex closest to the viewer is more prominent, as it should be. Figure 7(c) shows a 2x2x2 array of cubes, each divided into five tetrahedra, and rendered by linearly interpolated opacities. Notice the Mach Bands predicted in Figure 2. Figure 7(d) shows the same volume using the texture mapping for more accurate opacities, and is much improved.

The Sorting Algorithm

Most volume rendering algorithms use point sampling methods to calculate the color and intensity. Because the Shirley-Tuchman algorithm allows us to scan convert entire polyhedra very quickly, we needed to devise an efficient algorithm that would sort unstructured meshed elements in a back to front order. Our implementation will correctly sort arbitrarily shaped convex elements in a back to front order as long as there are no cycles or intersections in the data set. Each polyhedron can then be subdivided into a set of tetrahedra for rendering. If a convex mesh is structured so that cells meet on common faces, and this topological information is stored in an adjacency graph, then the adjacency graph can be used to produce a back-to-front sort (see [10] or [11]). However, we wanted to handle unstructured meshes where this data is unavailable. Such examples are sliding interfaces, where cells meet on only part of their faces, and non-convex meshes, such as those with cavities. Figure 3 shows such features in a mesh of a piston inside a cylinder. We extended the Newell, Newell and Sancha sort for polygons to correctly handle convex polyhedra. The sort will not perform subdivisions in the case of intersecting polyhedra or cycles, instead it will render them in an arbitrary order.

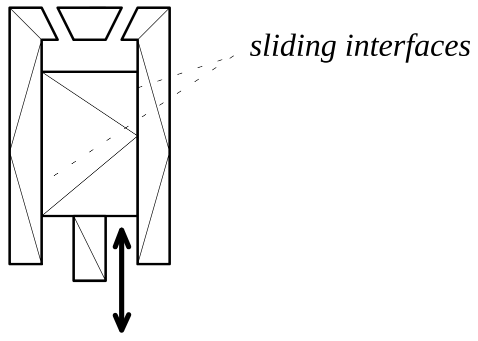


Figure 3.

This algorithm is a three dimensional extension of the Newell, Newell and Sancha painter's algorithm [5-8] and operates on the volumes after having performed all of the perspective transformation operations. Once the elements have been sorted in back-to-front order, they can be fed to the volume renderer for scan conversion and compositing, using the techniques described above.

There are three stages to the sorting process. The first applies all viewing transformations on the vertices to obtain the screen coordinates with a perspective corrected Z. The second obtains a rough sorting of the polyhedra based on the rearmost Z component of each element. Since we have applied the viewing transformation to all vertices and have scaled Z so as to correct for perspective, we would like to sort by increasing Z (the eye looks down the Z axis towards negative infinity in a right-handed coordinate system). In our implementation, this rough sort was obtained through an $O(n \log n)$ QuickSort. The third stage, or "fine tuning" of the sort, is a bit more complicated. However, like the painter's algorithm approach, it is also broken down into multiple steps with each one increasing in computational complexity, in hopes that a majority of the polyhedra will pass the earlier and less expensive tests.

The goal of the fine tuning is to find a separating plane between two polyhedra, P and Q, in order to determine whether or not P can be safely drawn before Q. The fine tuning process is broken down into five steps in order to efficiently find this separating plane. Given a list H of polyhedra roughly sorted by increasing Z coordinate of the rearmost vertex (called $Z_{rearmost}$), let polyhedron P be at the head of the list. P can be safely rendered if, for all polyhedra Q in the list H whose $Z_{rearmost}$ is less than (behind) P's $Z_{frontmost}$, the following function returns a value of True:

```
Test_Polyhedra(P,Q)
{
   if (P and Q do not have
       overlapping X extents) return True
     else if (P and Q do not have
         overlapping Y extents)
         return True
       else if (P is behind a
           back-plane of Q) return True
         else if (Q is in front of a
            front-plane of P) return True
           else if
              (Q!=EdgeIntersection(P,Q))
            return True
               else return False
}
```

The function `EdgeIntersection(P,Q)` returns the polyhedron which it determines to be in back. It makes this decision by looking for intersections between the edges of P's projection and the edges of Q's projection. If one is found, it finds the Z component of that intersection point for P and for Q, and returns the polyhedron whose $Z_{intersection}$ is farther from the eye. In the case that they are both equal, then we continue searching for intersections looking for an inequality between the two $Z_{intersection}$ components.

If the above function returns False, then polyhedra P and Q are considered to be in the wrong order and Q should be moved to the head of the list and the tests should be repeated with Q becoming the new P. It is possible that the list H could contain a cycle. For instance, if polyhedron A obscures B, and B obscures C, and C, in turn, obscures A, then there is no correct ordering for the polyhedra involved. Figure 4 illustrates a cycle for three polyhedra. The existence of a cycle is easily determined by tagging polyhedron Q before inserting it at the head of the list after the `Test_Polyhedra()` function fails. If Q has already been tagged, then a cycle exists and it will need to be addressed.

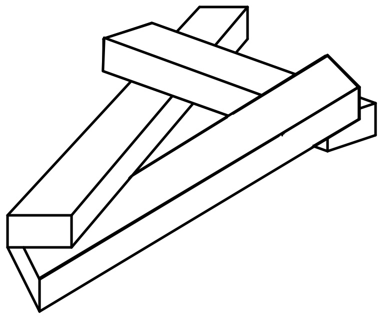


Figure 4.

If polyhedron P passes the tests for all polyhedra Q where $Q_{rearmost}$ is less than $P_{frontmost}$, then polyhedron P is free to be rendered; the tests have determined that P will not obscure any polyhedra which are considered to be in front of it. P is then shipped to the renderer and the next polyhedron in the list is chosen for the new P.

The first two tests check the bounding boxes of the two polyhedra in the X and Y plane. The main thrust of the third and fourth tests is to find a separating plane between P and Q. If such a plane exists, then P can safely be considered to lie behind Q. To simplify the third and fourth tests, we can mark each face of every polyhedron as being either a front-facing polygon (it faces the eye) or a back-facing polygon. This is easily determined because the algorithm stores an outward pointing normal for each face. Therefore, a simple query as to the sign of the Z component of a face's normal is enough to determine whether the face is front facing or not. A positive Z, in a right-handed coordinate system, is front facing. Otherwise it is back-facing. This pre-processing is all performed while reading in the meshed topology.

The third test then simplifies to testing whether all of P's vertices lie behind a plane defined by any one of Q's back-facing polygons. If this is true, then the face under consideration forms a separating plane between P and Q and therefore we can conclude that P is behind Q. Performing this test is a matter of making sure that for at least one back-facing polygon of Q, the sign of $f(x_j, y_j, z_j)$ for all vertices *j* in P is non-negative for that particular face of Q. The plane equation, *f*, is based on the outward pointing normals for that face. If this test fails, then the algorithm will proceed to the fourth test and try to determine whether the plane specified by a front-facing polygon belonging to P separates P from Q.

This fourth test is very similar to the third test. In determining whether Q lies entirely in front of P, one must make sure that for at least one front-facing polygon of P, $f(x_j, y_j, z_j)$ is positive for all vertices *j* in polyhedron Q. This time, f is the plane equation for a front-facing polygon of P, again based on outward pointing normals. If this test passes, then Q lies entirely in front of at least one of the front-facing polygons of P and it can be concluded that P lies behind Q.

The fifth test, `EdgeIntersection()`, returns either the index of the polyhedra which is in back, or an error condition if it cannot detect any intersecting edges. The two cases where this test can fail are shown in Figure 5. As we will see, this does not jeopardize the correctness of our algorithm.

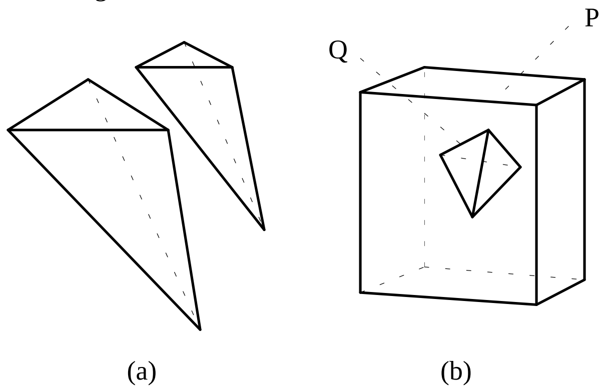


Figure 5.

The illustrations (a) and (b) in Figure 5, which both represent screen projections, both fail the `EdgeIntersection()` function because neither have intersecting edges in their projections. However, in case (a) the order in which the two tetrahedra are rendered makes no difference since they are completely disjoint in the screen projection and therefore an error condition can correctly be treated as if polyhedron P were in front of polyhedron Q. On the other hand, this is not necessarily the situation in case (b). We can rest assured that this will never cause a sorting glitch because the front face of the brick (assuming the tetrahedron is in front of the brick) is a front-facing separating plane and would have been caught in the fourth test. This fifth test is a more efficient alternative to the linear programming method proposed by Newell [6]. If the fifth test fails, then polygon Q should be moved to the front of the list and the whole process should be repeated.

With the exception of the fifth, these tests are very easy to perform. When reading in the topological data-set, one must store the plane equation coefficients, with respect to an outward pointing normal, in the polyhedral database. From these pre-computed coefficients, determining which side of a face a point *j* lies is as simple as finding the sign of $ax_j + by_j + cz_j + d$.

In the case that all of the tests fail and we have a cycle, the program will render first whichever of P and Q has a $Z_{frontmost}$ further from the eye.

Non-planar Faces

The algorithm described works correctly for convex polyhedra with planar faces and no cycles or intersections. Unfortunately, it is quite possible, in finite element codes, for the faces to skew slightly yielding non-planar faces. Fortunately, the faces will be mostly planar because highly non-planar faces can lead to instabilities in the modeling code. Figure 6 illustrates an exaggeration of what could possibly happen. Even if the face were mildly non-planar, it is still enough to cause the tests to fail. To accommodate slightly non-planar faces, we have introduced an error tolerance δ.

In order to sort convex polyhedra with non-planar faces as shown in Figure 4, the algorithm first calculates an average outward pointing normal, (*a*,*b*,*c*), for each face. This is done using Newell's method as follows [8,9]:

$$a = \sum_{i=1}^{n} (y_i - y_j)(z_i + z_j)$$

$$b = \sum_{i=1}^{n} (z_i - z_j)(x_i + x_j)$$

$$c = \sum_{i=1}^{n} (x_i - x_j)(y_i + y_j)$$

where: *j*=(*i*+1) mod *n*
and *n* is the number of vertices per polygon

The last coefficient of the plane equation, d, can be calculated by picking some point on the average plane. We chose the center of gravity of the face for this point as follows:

$$\frac{1}{n}\left(\sum_{i=1}^{n} x_i, \sum_{i=1}^{n} y_i, \sum_{i=1}^{n} z_i\right)$$

To determine on which side of a plane a point lies, an error tolerance is used. This is needed because with non-planar faces the algorithm could return vertex *a* of polyhedron Q as being contained inside of P, which would ultimately result in a cycle (see Figure 6). This is not the case. In fact, if vertex *a* were actually touching a plane of polyhedron P, machine round-off might place *a* on the wrong side of that face which, again, would result in a cycle. Therefore a tolerance, δ, is used to represent an acceptable distance from a vertex to a face. In other words, the third and fourth tests should consider vertex *a* to be on the outside of a face (the plane equation evaluated at point *a* should yield a non-negative value) if point *a* is within δ units away from the plane under consideration, regardless of which side of the face point *a* actually lies. We can rationalize the existence of this δ tolerance as follows: if a corner of polyhedron Q happens to intersect a planar face of polyhedron P by the amount δ, for a suitably small δ, the visual impact will be minimal, if perceptible at all. Our implementation uses a unique δ for each face, based on the maximum deviation of a vertex from its corresponding average plane.

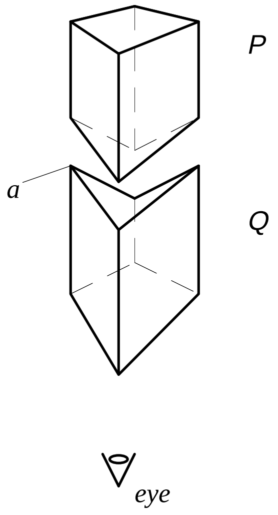


Figure 6.

Discussion

The fine-tuning sort described runs in $O(n^2)$with respect to the number of polyhedra sorted. However, this quadratic running time is an upper bound and would only be found in the most pathological cases where all polyhedra have overlapping Z extents. The average running times for normal data sets should be lower. While the first and second tests run in constant time, the third and fourth tests run in $O(F_iE_j)$ and $O(E_iF_j)$ time where E_i and E_j correspond to the number of edges for polyhedra *i* and *j*, respectively, and F_i and F_j are the number of faces. The fifth test runs in $O(E_iE_j)$ Again, this is a worst case running time and it should be substantially better in practice since the function terminates once a suitable intersection in the two projections is found.

The algorithm was implemented in C++ and has been used to sort those primitives found in the SGI Explorer pyramid type. The volume primitives are all subclasses of a general *primitive* C++ class. These subclasses are as follows: the tetrahedra, pyramid, prism, wedge and brick. We can easily extend the system to include others.

Table 1 shows some timing data using the complete sort on an SGI Indigo2™. As contrast, the QuickSort can sort 24,000 elements in 4 seconds, and 157,000 elements in 27 seconds.

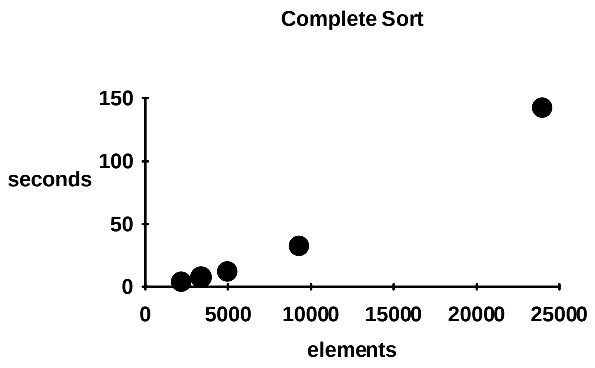


Table 1.

Figure 7(e) illustrates the results of the sort, with the Williams and Max color integration, on the "blunt fin".

We present no new approaches to cycle breaking. If a cycle is detected during the sorting, then the polyhedron with the vertex farthest from the eye would be removed from the list and rendered. The most common form of a cycle the algorithm would detect in a data set would probably be two non-planar faced polyhedra "intersecting" each other. However, the δ overlapping tolerance should eliminate most of these situations. The traditional, but slower method for removing cycles, such as the type illustrated in Figure 4, would be to pass one or more cutting planes through the offending polyhedra.

Conclusion

This paper presents extensions to the Shirley-Tuchman algorithm for compositing colored elements with hardware assisted texture mapping. We have also presented extensions to the Newell, Newell, and Sancha sort for use with unstructured data. For quick interaction or still frames, QuickSorting alone is adequate. For a final animation, the full sort is necessary because popping will become apparent if the rendering order suddenly becomes incorrect.

Acknowledgments

We would like to thank Roger Crawfis for all of his help and suggestions. We would also like to thank the people at Silicon Graphics for their advice and help in creating our Explorer modules.

This work was performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

References

[1] Shirley, P. and A. Tuchman, "A Polygonal Approach to Direct Scalar Volume Rendering", *Computer Graphics*, Vol. 24, No. 5 (November 1990), pp 63-70.

[2] Max, N, B. Becker, and R. Crawfis, "Flow Volumes for Interactive Vector Field Visualization", *Proceedings of Visualization '93*, (October 1993), pp. 19-25.

[3] Kirk, D. and D. Voorhies, "The Rendering Architecture of the DN10000VS", *Computer Graphics*, Vol. 24, No. 4, (August 1990), pp 299-307.

[4] Williams, P. and N. Max, "A Volume Density Optical Model", *1992 Workshop on Volume Visualization*, Association for Computing Machinery, New York (1992) pp. 61-68.

[5] Newell, M. E., R. G. Newell, and T. L. Sancha, "A Solution to the Hidden Surface Problem." *Proceedings of the ACM National Conference 1972*, pp. 443–450.

[6] Newell, M. E. "The Utilization of Procedure Models in Digital Image Synthesis", Ph.D. Thesis, University of Utah, 1974 (UTEC-CSc-76-218 and NTIS AD/A 039 008/LL).

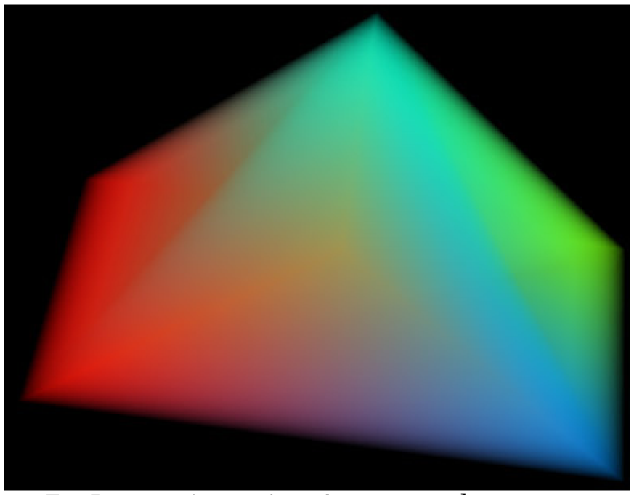
[7] Foley, J., A. van Dam, S. Feiner, J. Hughes, *Computer Graphics Principles and Practice, 2nd Edition,* Addison-Wesley, Reading, Massachusetts, 1990.

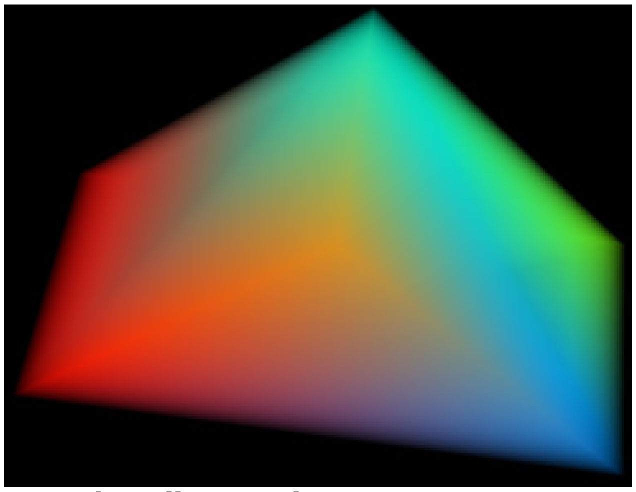
[8] Rogers, David F., *Procedural Elements for Computer Graphics,* McGraw-Hill, New York. 1985.

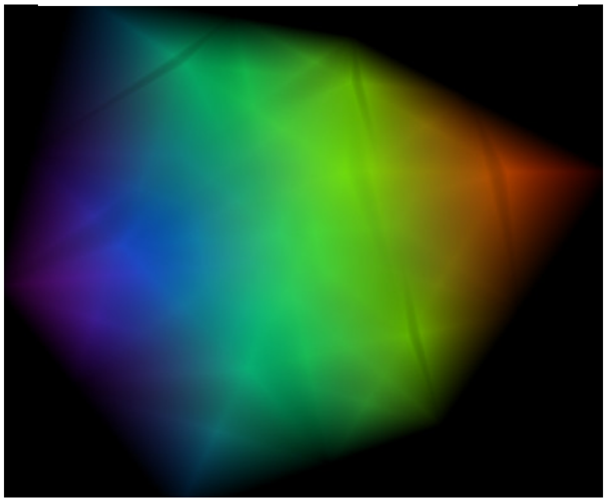
[9] Sutherland, I. E., R. F. Sproull, and R. A. Schumaker, "A Characterization of Ten Hidden-Surface Algorithms", *Computing Surveys*, Vol. 6, 1974, pp. 1-55.

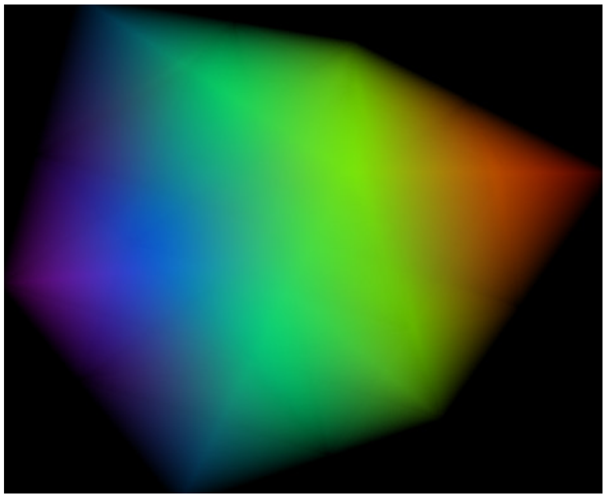
[10] Max, N., P. Hanrahan, and R. Crawfis, "Area and Volume Coherence for Efficient Visualization of 3-D Scalar Functions", *Computer Graphics*, Vol. 24, No. 5 (November 1990), pp. 27-33.

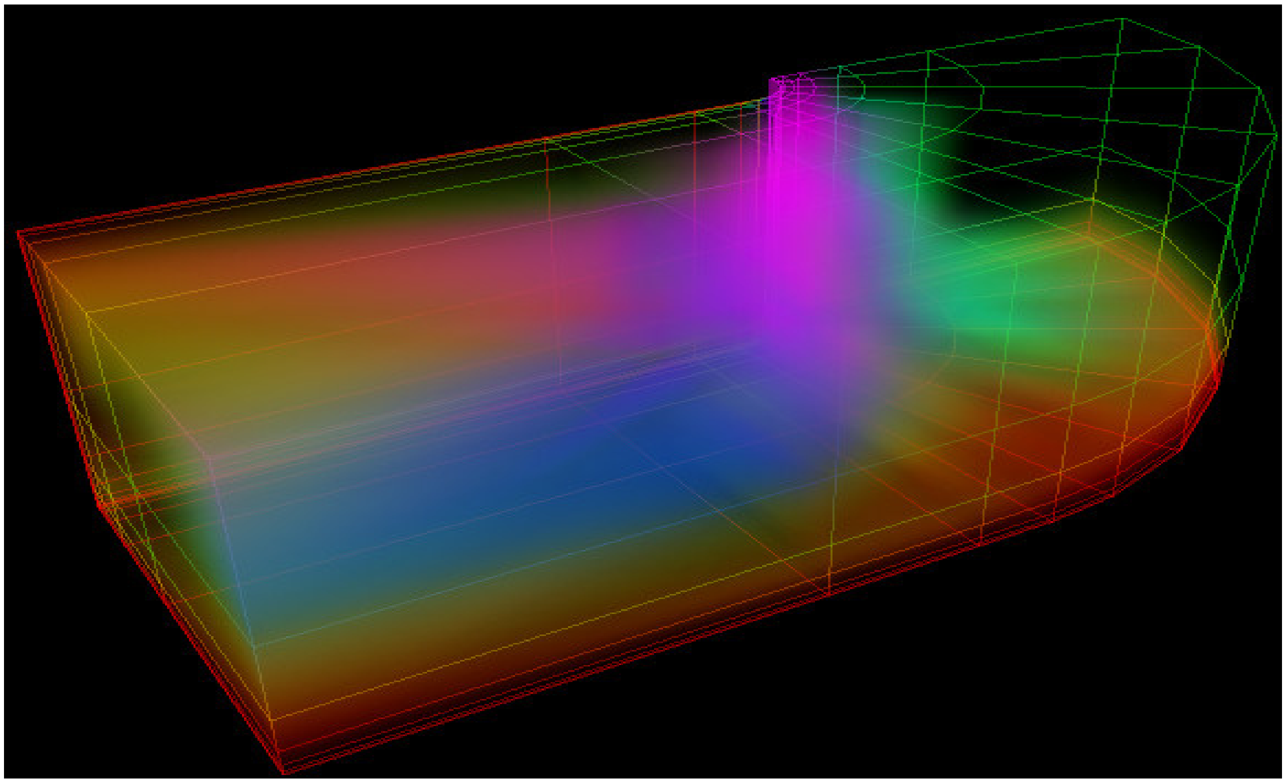
[11] Williams, P. "Visibility Ordering of Meshed Polyhedra", *ACM Transactions on Graphics*, Vol. 11, No. 2, (April 1992), pp.103-126.


7a. Integration using Average color.


7b. Williams' color integration.


7c. No texture mapping.


7d. With texture mapping.


7e. Blunt fin with 440 sorted brick elements, using texture mapping.